Chapter 2

Variant's assessment

RQ2. To what extent the generated variants are different ?

In this chapter, we investigate to what extent the artifically created variants are different. We propose a methodology to compare the program variants both statically and during runtime. Besides, we present a novel study on code preservation, demonstrating that the code transformations introduced by CROW are resilient to later compiling transformations during machine code generation. We evaluate the variant's preservation in both existing scenarios for WebAssembly, browsers and standalone engines.

2.1 Metrics

In this section we propose the metrics used along this chapter to answer RQ2. We define the metrics to compare an original program and its variants statically and during runtime. Besides, we proposed the metrics to compare program variants preservation.

2.1.1 Static

To measure the static difference between programs, we compare their bytecode instructions using a global alignment approach. In a previous work of us we empirically demonstrated that programs semantic can be detected out of its natural diversity [?] . We compare the WebAssembly of each program and its variant using Dynamic Time Warping (DTW) [**?**]. DTW computes the global alignment between two sequences. It returns a value capturing the cost of this alignment, which is actually a distance metric. The larger the DTW distance, the more different the two sequences are.
TODO Add and example here ?

Metric 1 *dt_static: Given two programs P_X and V_X written in X code, dt_static(P_X, V_X), computes the DTW distance between the corresponding program instructions for representation X.*

A dt_static(P_X, V_X) of 0 *means that the code of both the original program and the variant is the same, i.e., they are statically identical in the representation X. The higher the value of dt_static, the more different the programs are in representation X.*

Notice that for comparing WebAssembly programs and its variants, the metric is the instantiation of dt_static with $X = WebAssembly$.

2.1.2 Program traces and execution times

We measure the difference between programs at runtime by evaluating their execution trace, at function and instruction level. Also, we include the measuring of the execution time of the programs. Besides, we compare their execution times.

Metric 2 *dt_dyn: Given a program P, a CROW generated variant P' and T a trace space ($T \in \{Function, Instruction\}$) dt_dyn(P,P',T), computes the DTW distance between the traces collected during their execution in the T space. A dt_dyn of* 0 *means that both traces are identical.*

The higher the value, the more different the traces.

Metric 3 *Execution time: Given a WebAssembly program P, the execution time is the time spent to execute the binary.*

2.1.3 Variants preservation

The last metric is needed because WebAssembly is an intermediate language and compilers use it to produce machine code. For program variants, this means that compilers can undo artificial introduced transformations, for example, through optimization passes. When a code transformation is maintained from the first time it is introduced to the final machine code generation is a preserved variant.

Part of the contributions of this thesis are our strategies to prevent reversion of code transformations. We take engineering decision regarding this in all the stages of the CROW workflow. We disable all optimizations inside CROW in the generation of the WebAssembly binaries. This prevents the LLVM toolchain used to remove some introduced transformations. However, the LLVM toolchain applies optimizations by default, such as constant folding or logical operations' normalization. As we illustrate previously, these are some transformations found and applied by CROW. We modified the LLVM backend for WebAssembly to avoid this reversion during the creation of Wasm binaries. This phenomenon is sometimes bypassed by diversification studies when they are conducted at high-level. As another contribution, we conduct a study on preservation for both scenarios where Wasm is used, browsers and standalone engines. In

The final metric corresponds to the preservation study. We compare two programs to be different under the WebAssembly representation and under the machine code representation after they are compiled through a collection of selected WebAssembly engines. We use two instances of Metric 1 for two different code representations, WebAssembly and x86. The key property we consider is as follows:

Property 1 *Preservation: Given a program P and a CROW generated variant P', if dt_static(*P_{Wasm}*,* P'_{Wasm}*) > 0 and dt_static(*P_{x86}*,* P'_{x86}*) > 0* $\implies$ *both programs are still different when compiled to machine code.*

If the property fits for two programs, then the underlying compiler does not remove the transformations made by CROW. Notice that, this property only makes sense between variants of the same program, including the original.

TODO Improve this !

Metric 4 *Preservation ratio*

Given a program P and a corpus of variants V generated by CROW from P.

$$preservation\_ratio = \frac{|v_1, v_2 \in V \cap \{P\}, \forall v_1, v_2 \ ensuring\ Property\ 1|}{|V \cap \{P\}|^2}$$

Notice that Metric 4 implies a pairwise comparison between all variants and the original program.

2.2 Evaluation

To answer RQ2 we use the same corpora proposed and evaluated in chapter 1, **CROW prime** and **MEWE prime**.

TODO One paragraph per metric

We compare the sequence of instructions of each variant with the initial program and the other variants. We obtain two DTW distance values for each program-variant pair: one at the level of WebAssembly code and the another one at the level of x86 code. Metric 1 below defines these metrics.

To answer **??** we compute Metric 2 for a study subject program and all the unique program variants generated by CROW in a pairwise comparison. The pairwise comparison allows us to compare the diversity between variants as well. We use

SWAM[1] to collect the stack operation traces. SWAM is a WebAssembly interpreter that provides functionalities to capture the dynamic information of WebAssembly program executions including the stack operations. We compute the DTW distances with STRAC [**?**].

The engines used are listed in Table 2.1. We only measure the difference in x86 after the WebAssembly code is compiled to the machine target. This decision is not arbitrary, according to the study of **TODO** Paper on binary diff survey , any conclusion carried out by comparing two program binaries under a specific target can be extrapolated to another target for the same binaries.

Name	Properties
V8 [?]	V8 is the engine used by Chrome and NodeJS to execute JavaScript and WebAssembly. **TODO** Explain compilation process
wasmtime [?]	Wasmtime is a standalone runtime for WebAssembly. This engine is used by the Fastly platform to provide Edge-Cloud computing services. **TODO** Explain compilation process

Table 2.1: Wasm engines used during the diversification assessment study. The table is composed by the name of the engine and the description of the compilation process for them.

2.3 Results

2.3.1 Static

2.3.2 Dynamic

2.3.3 Preservation

2.4 Conclusions

[1]`https://github.com/satabin/swam`